Cairo University

Faculty of Computing and Artificial Intelligence

CS321 - Algorithms Analysis and Design

The Arcadia Engine

CAPSTONE IN ALGORITHMS & DATA STRUCTURES

1. Background / Real-World Context

Scenario: Your team has been hired by *Arcadia Games* to build the backend engine for their new MMORPG (Massively Multiplayer Online Role-Playing Game). The game requires a high-performance system to handle player data, inventory management, world navigation, and server task scheduling.

Instead of writing isolated scripts, you will build the **ArcadiaEngine** library. The engine consists of four subsystems:

1. **The Registry:** Handles player lookups and leaderboards (Hashing & Skip Lists).
2. **The Inventory System:** Manages loot distribution and storage (Dynamic Programming).
3. **The Navigator:** Manages travel routes and safe paths (Graphs).
4. **The Kernel:** Manages server CPU tasks (Greedy Algorithms).

2. Integrated Tasks

Part A: The Registry (Data Structures)

Context: The game needs to store player profiles and maintain a live leaderboard.

1. **Player Lookup (Hashing):**

   Implement a *PlayerTable* using a **Hash Table**.

   - **Collision Resolution:** You **must** use **Double Hashing**.
   - **Table Size:** The table size is fixed at **101**.
   - **Storage:** You may use a standard array or std::vector **ONLY** if initialized with this fixed size. Dynamic resizing (push_back) is prohibited.
   - **Behavior:**
     - If a collision occurs, use the double hashing step to probe for an empty slot.
     - If the table is full (after probing), throw the error "Table is full". Do **not** resize.

2. **Live Leaderboard (Skip List):**

   Implement a Leaderboard class using a **Skip List**.

   - **Operations:** Support $O(log\ n)$ insertion and searching by score.
   - **Ordering:** The list must be sorted by **Score (Descending)**.
   - **Tie-Breaking (Crucial):** If two players have the same score, sort them by **Player ID (Ascending)**.
     - *Visual Example:* Player A (ID 10, Score 500) must appear **before** Player B (ID 20, Score 500).
   - **Deletion Exception:** For removePlayer(int playerID), you are **allowed** to use an $O(N)$ linear scan to find the node by ID, then perform the standard $O(log\ n)$ pointer updates to delete it.
   - **Function:** getTopN(int n) must retrieve the top N player IDs efficiently.

3. **Auction House (Red-Black Tree):**

   Implement an AuctionTree using a **Red-Black Tree** to store items for sale.

   - **Ordering:** Ordered by **Price**.
   - **Duplicate Prices:** You must handle duplicate prices. Suggestion: Use a composite key comparison (Price, then ID) to ensure uniqueness in the BST.
   - **STL Restriction:** You are **strictly forbidden** from using std::map, std::unordered_map, or any auxiliary STL container to map IDs to nodes.
   - **Deletion Complexity:** Since deleteItem(int itemID) requires deleting by a non-primary key (ID), you are **allowed** to perform an $O(N)$ traversal to find the node, followed by the standard $O(log\ n)$ Red-Black deletion fix-up.

Part B: The Inventory System (Dynamic Programming)

Context: Players find loot and need to organize it.

1. **Loot Splitting (Partition Problem):** Two guild leaders want to split a bag of gold coins as evenly as possible. Given a list of coin values, minimize the difference between the two sums.

Input	Output	Explain

n=3
coins = {1,2,4}
	1	Best split: {4} vs {1,2}. Difference: 4-3 = 1

n=2
coins = {2,2}
	0	Perfect split: {2} vs {2}. Difference: 0

n=4
coins = {1,5,11,5}
	0	Best split: {11} vs {1,5,5}. Both sum to 11

2. **Inventory Packer (Knapsack):** A player has a backpack with weight capacity W. Given a list of items with weight w_i and value v_i, maximize the total value they can carry. (Note: This combines the logic of the "Bank Robber" and "Diver" problems).

Input	Output	Explain

capacity=3
items = {{1,10}, {2,15}, {3,40}}
	40	Select item 3 (weight=3, value=40) only

capacity=5
items = {{1,10}, {2,15}, {3,40}}
	55	Select items 1 and 3 (total weight=4, value=50) OR items 2 and 3 (total weight=5, value=55)

capacity=10
items= {{1,10}, {2,20}, {3,30}}
	60	All items fit (total weight=6). Total value=60

3. **Chat Autocorrect (String DP):** The chat system is buggy. Given a received string (where 'w' might be 'uu' and 'm' might be 'nn'), calculate the number of possible original strings modulo $10^9 + 7$.

Input	Output	Explain
"uu"	2	Possible originals: "uu" (no substitution) or "w" (reverse substitution)
"uunn"	4	Possible: "uunn", "wnn", "uum", "wm"

Part C: The Navigator (Graph Algorithms)

Context: Players need to traverse the game world consisting of N cities and M roads.

1. **Safe Passage (Path Existence):** Determine if a valid path exists between a source city and a destination city given a set of bidirectional edges.

Input	Output	Explain

n=3
edges = {{0,1}, {1,2}}
source=0
dest=2
	true	Path exists: 0→1→2

n=4
edges= {{0,1}, {2,3}}
source=0
dest=3
	false	No path connects 0 and 3 (disconnected components)

n=1
edges = {}
source=0
dest=0
	true	Source equals destination

2. **The Bribe (Min Cost Path/MST variant):** Roads are guarded by bandits requiring specific gold/silver payments. Given the exchange rate between gold/silver and "Olympian Tugriks," find the minimum cost to ensure connectivity or specific passage (Variation of the "Kingdom of Olympia" problem).

Input	Output	Explain

n=3
m=3
goldRate=1
silverRate=1
	15	Roads: {{0,1,10,0}, {1,2,5,0}, {0,2,20,0}}
		Costs: Road1=10, Road2=5, Road3=20
		MST picks roads 2,1 → Total: 15

3. **The Teleporter Network (All-Pairs Shortest Path):** The world has road lengths that are powers of two. Calculate the sum of minimum distances between all pairs of cities (binary representation required).

Input	Output	Explain
n=3	"110"	Distances: 0-1=1, 1-2=2, 0-2=3.

roads= {{0,1,1}, {1,2,2}}		Sum=6. Binary: 110

n=2
roads = {{0,1,4}}
	"100"	Distance: 0-1=4.
		Sum=4.
		Binary: 100

n=3
roads = {{0,1,2}, {0,2,8}}
	"1010"	Distances: 0-1=2, 0-2=8, 1-2=∞ (disconnected).
		Sum=10.
		Binary: 1010

Part D: The Kernel (Greedy Algorithms) Server Job Scheduler: The server processes tasks 'A'-'Z'. Tasks of the same type require a cooling interval n. Calculate the minimum CPU intervals to finish all tasks.

Constraints:

- 1 <= tasks.length <= 104
- tasks[i] is an uppercase English letter.
- 0 <= n <= 100

Input	Output	Explain

tasks={'A','A','B'}
n=2
	4	Schedule: A → B → idle → A.
		Total: 4 intervals

tasks={'A','A','A'}
n=2
	7	Schedule: A → _ → _ → A → _ → _ → A.
		Total: 7 intervals

tasks={'A','B','C'}
n=2
	3	All unique tasks, no cooling needed.
		Total: 3 intervals

tasks={'A','A','A','B','B','B'}
n=2
	8	Schedule: A → B → idle → A → B → idle → A → B.
		Total: 8 intervals

3. FAQ

General & Submission

Q: Can I modify ArcadiaEngine.h?

A: No. You must not modify ArcadiaEngine.h under any circumstances. Any changes to the header file will cause the auto-grader to fail your submission, resulting in a grade of zero.

Q: Can I add my own helper functions or structs?

A: Yes. You have full freedom to edit ArcadiaEngine.cpp. You are encouraged to add private helper functions, structs, or classes inside ArcadiaEngine.cpp to keep your code organized.

Q: Can I use STL containers (like $\boldsymbol{std::map}$ or $\boldsymbol{std::set}$) inside my Data Structures (Part A)?

A: No. Using $std::map$, std::unordered_map, $std::set$, or $std::unordered\_set$ inside the internal implementation of the Registry (Part A) is **strictly forbidden** and will result in a zero grade for that component.

Q: Is $\boldsymbol{std::vector}$ allowed? A:

- **In Part A (Data Structures):** You may use $std::vector$ *only* for storage (e.g., storing forward pointers in a Skip List node or the fixed-size backing array for the Hash Table).
- **In Parts B, C, D:** You are free to use $std::vector$ and other standard containers as needed.

Part A: The Registry (Data Structures)

1. Hashing (Player Table)

Q: What should the Hash Table size be?

A: The table size must be fixed at **101**.

Q: Can I resize the Hash Table if it gets full?

A: No. Dynamic resizing is not allowed. You must initialize your array or vector with a fixed size of 1018. If the table is full after probing, simply output/return an error message “Table is full” without double quotation (e.g., Table is full).

Q: How do I handle collisions?

A: You must use **Double Hashing**. Do not use separate chaining or linear probing alone.

2. Skip List (Leaderboard)

Q: How should the Skip List be ordered?

A: It must be ordered primarily by Score (Descending).

- **Tie-Breaker:** If scores are equal, order by **Player ID (Ascending)**.
- **Visual Example:** Player A (ID 10, Score 500) must appear *before* Player B (ID 20, Score 500) because $10 < 20$.

Q: Doesn't searching for a player by ID (for removePlayer) take $O(N)$?

A: Yes, and that is allowed. Since the list is sorted by Score, searching by ID requires a linear scan. For this assignment, an $O(N)$ search to find the node followed by $O(log\ N)$ deletion logic is acceptable.

3. Red-Black Tree (Auction House)

Q: The Auction Tree is ordered by Price, but deleteItem uses itemID. Can I use a map to find the node efficiently?

A: No. Using an auxiliary $std::map$ is forbidden.

- You are allowed to perform an $O(N)$ traversal (linear search) to find the node by itemID.
- Once found, you must perform the standard Red-Black Tree deletion, which is O(log N).

Q: How do I handle items with duplicate prices?

A: The Red-Black Tree must support duplicate prices.

- **Suggestion:** Use a composite key for comparison: if (Price A == Price B), compare their IDs to decide left/right placement. This ensures every node is unique in the tree structure.

Parts B, C, & D

Q: What is the return value for optimizeLootSplit (Part B)?

A: Return the **minimum difference** (as an integer) between the sums of the two subsets[15].

- *Example:* Coins {1, 2, 4} => Split {4} vs {1, 2} => Difference $4 - 3 = 1$.

Q: What is the format of items in maximizeCarryValue (Knapsack)?

A: Each item is provided as a pair {weight, value}.

Q: What does minBribeCost calculate (Part C)?

A: It calculates the minimum total cost to connect **ALL** cities, which is the **Minimum Spanning Tree (MST)** problem.

Q: For the Job Scheduler (Part D), does n=0 mean no cooling time?

A: Yes. If n=0, tasks can be executed immediately one after another without idle intervals19.

Q: What are the expected time complexities?

A: Hash table $O(1)$ avg, Skip list $O(log\ n)$, RB tree $O(log\ n)$,

Knapsack $O(n \times W)$, Loot Split (Partition) $O(n \times sum)$, String $O(n)$,

Path Exists (BFS/DFS) $O(V + E)$, MST $O(E\ log\ V)$, **All-Pairs Shortest Path** – APSP (Teleporter) $O(V^3)$, Scheduler $O(n\ log\ n)$

4. Implementation & Testing Guidelines

1. File Restrictions (ArcadiaEngine.h)

- **STRICT PROHIBITION:** You are **NOT** allowed to modify, delete, or add anything to the ArcadiaEngine.h header file.
- **Consequence:** The auto-grader relies on this specific header. Any alteration to this file will cause compilation errors with the grading suite, resulting in an automatic **zero** for the assignment.

2. Implementation (ArcadiaEngine.cpp)

- Your entire implementation must be done inside ArcadiaEngine.cpp.
- You must implement all functions marked with TODO or left empty[4].
- **Helper Functions:** You have full freedom to add **private** helper functions, structs, or classes inside ArcadiaEngine.cpp to organize your logic (e.g., a private searchByID helper for the RB Tree).

3. Testing Your Code

- We have provided a file named main_test_student.cpp.
- **Action:** You should compile and run your code against this file to verify that your basic logic works as expected.
- **Warning:** Passing main_test_student.cpp does **not** guarantee a full mark.
  - This file only covers the "Happy Path" (basic scenarios).
  - **TA Testing:** The teaching assistants will use a much more comprehensive test suite (including main_test.cpp) that checks for:
    - **Edge Cases:** (e.g., empty inputs, single nodes, maximum values).
    - **Performance:** (e.g., ensuring operations are $O(log\ n)$ or $O(1)$ as required).
    - **Memory Leaks:** (e.g., proper deletion of nodes).
- **Advice:** Do not stop at the provided tests. Write your own additional test cases to handle extreme scenarios.

5. Submission Guidelines:

- Teams must consist of 4 to 5 members.

    - A penalty will be imposed for groups with fewer than 4 or more than 5 members will incur point deductions.

- Beside pdf there are two files *ArcadiaEngine.h* and *ArcadiaEngine.cpp*.

- Do not modify *ArcadiaEngine.h* under any circumstances.

    - Any alterations to this file will result in automatic test case **failures and a grade of zero**.

- Implement TODO and empty function in *ArcadiaEngine.cpp*.

- You may add private helper functions or struct in *ArcadiaEngine.cpp*.

- All team members must understand all parts of the project.

    - if there are team member didn’t understand his/her work and his/her teammate work He / She will lose points

- No late submissions are allowed.

- Cheating is NOT tolerated by any means.

- Only one team member has to submit one compressed file (ZIP or RAR format) following the naming convention:

Assignment02_ID1_ID2_ID3_ID4 _ID5.zip/rar

    - if naming convention is wrong you will lose points

- A penalty will be imposed for violating any of the assignment rules or missing any deliverable.

- Cheaters will get ZERO and no excuses will be accepted as per the attached “Plagiarism Scope” document.

- TAs will grade the assignment out of 100, but this score may be adjusted later for scaling purposes.

- **Deadline 2025-12-18**

6. Deliverables

Team Submission (Compressed File): Assignment02_ID1_ID2_ID3_ID4_ID5.zip/rar

The submission must include:

1. ArcadiaEngine.cpp: Complete implementation of all required classes
2. TeamInfo.txt: Full names and student id of all team members

Cairo University

Faculty of Computing and Artificial Intelligence

CS321 - Algorithms Analysis and Design

Contents